COMP0037 ASSIGNMENT 2

Group: `Group L`

March 31, 2020

1 Reactive Planner

1.1 Reactive Planning System

A reactive planning system works like that: The planner makes a trajectory using the latest world model and the free space assumption. And the robot constantly perceives the environment and updates the world map as it moves. When the original path has become blocked by an obstacle, it reacts by planning a new set of actions and plans. See the block diagram in Fig. 1.

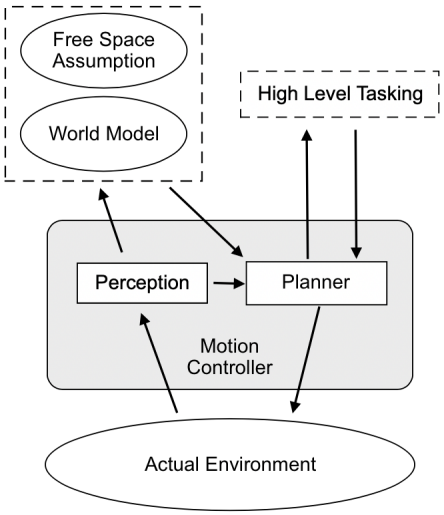


Fig. 1. Reactive Planning System

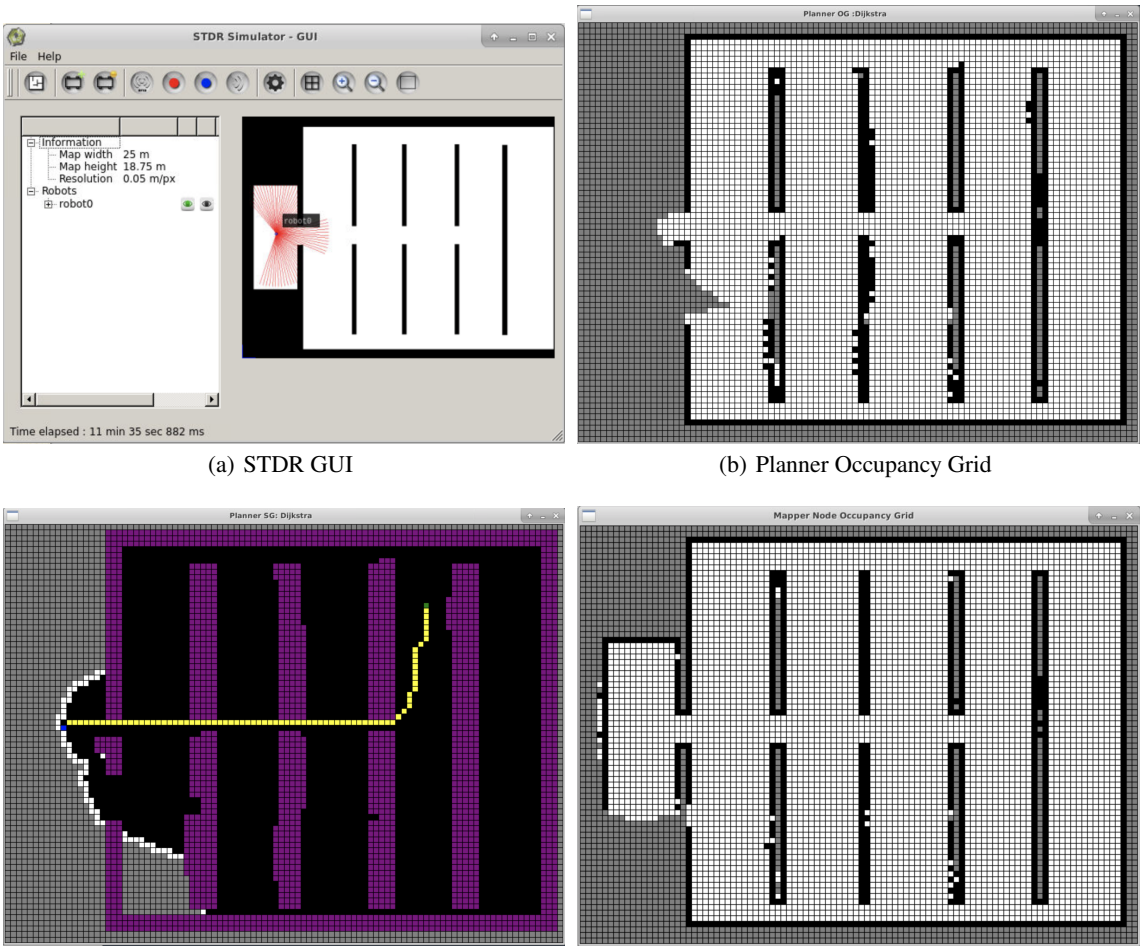
1.2 Our Implementation

We implement the reactive planning system by complete the function *checkIfCurrentPathIsStillGood* in the *ReactivePlannerContoller.py*. The code is shown in Fig. 2.

```
def checkIfPathCurrentPathIsStillGood(self):
    for waypoint in self.currentPlannedPath.waypoints:
        if self.planner.searchGrid.getCellFromCoords(waypoint.coords).label == CellLabel.OBSTRUCTED:
            self.controller.stopDrivingToCurrentGoal()
```

Fig. 2

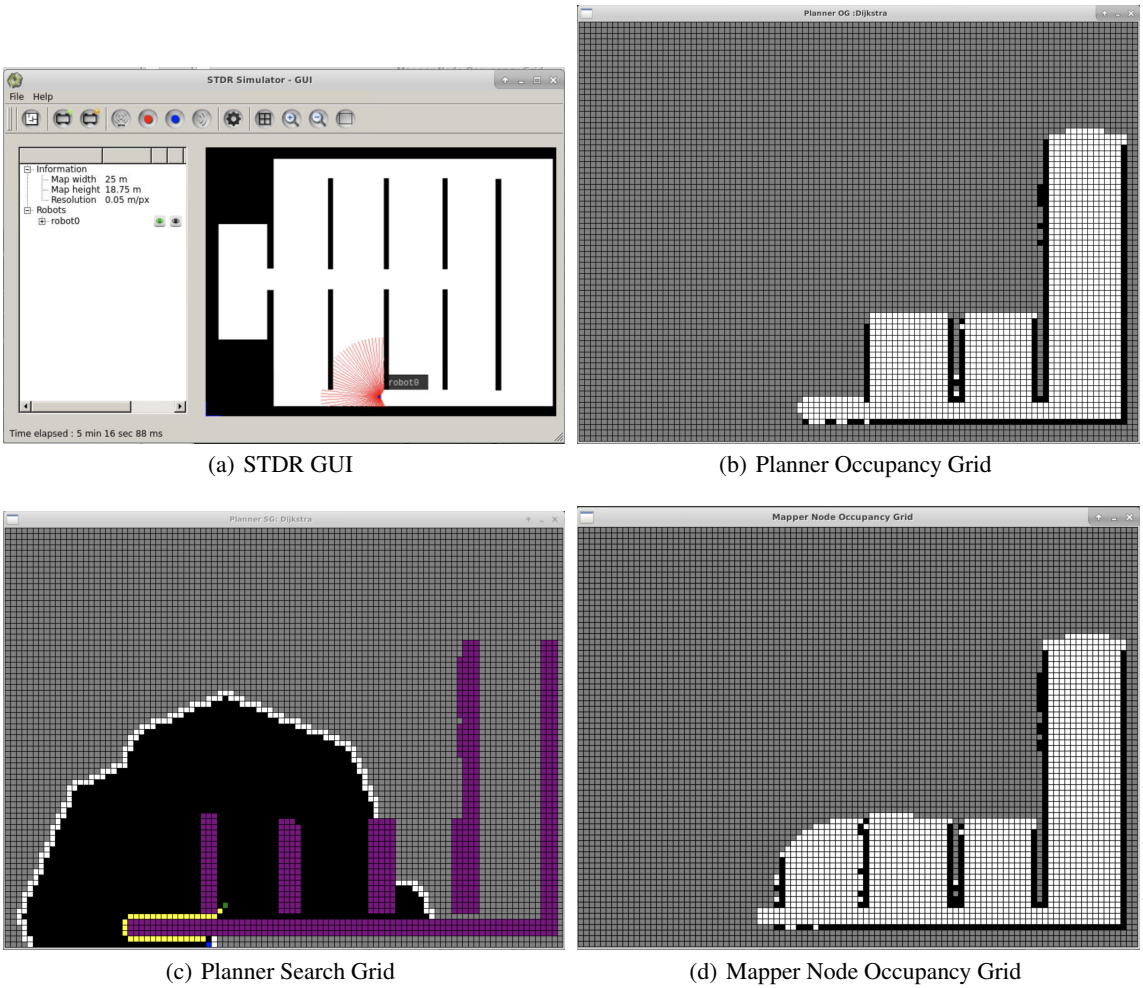
The result is shown in Fig.3 and Fig.4.



(a) STDR GUI (b) Planner Occupancy Grid

(c) Planner Search Grid (d) Mapper Node Occupancy Grid

Fig. 3. Result on the first launch script



(a) STDR GUI (b) Planner Occupancy Grid

(c) Planner Search Grid (d) Mapper Node Occupancy Grid

Fig. 4. Result on the second launch script

1.3 Approach for Improving the Performance

One approach for improving the performance is using a more efficient global planner to reduce the computational cost associated with re-planning. D*-Lite is one such algorithm that caches previous searches and update them only when we need to. It make use of the backwards search. The graph additionally contains a consistency condition. When the graph changes, such as due to an obstacle, the consistency condition is violated. A priority queue is then used to sort the effect of the failed condition on the path. The highest priority updates are committed first.

2 Frontier-Based Exploration System

2.1 Frontiers

A frontier is a cell which its state is known while it is adjacent to a cell whose state is not known. Frontier cells define boundary between open space and the uncharted territory. In frontier-based exploration, the robot moves to the boundary to gain the newest information about the world.

Two methods for identifying frontiers are wave front detection and fast frontier detection.The wave front detection explores the map based on the map that has already been explored. It searches the frontiers using depth-first search, starting from the robot initial location. Once a cell is encountered that looks like a frontier, it pauses the search and traces along all the frontier cells. The latter relies on the newly collected sensor information. The sensor data is used to create a contour, which then be separated into frontier and non-frontier segments. The frontier segments are managed in a database to make them persistent. When data becomes available, the frontiers will be split or merged deleted. One heuristic for choosing next waypoint is picking the closest frontier to the robot. Another one is picking the largest frontier cell.

2.2 The Exploration Algorithm Provided

The method provided finds frontiers by traverse all the cells on the occupancy grid each time. Once a frontier is found, it checks if it is in the blacklist. If not, it then calculates the distance between the cell and the centre of the leftmost boundary and see if its distance is smaller than the current smallest value. Finally, the frontier with the smallest distance (to the centre of the left boundary of the grid) is chose to be the next destination.

This method is extremely slow since the standard cell to compare the distance is the centre of the left boundary and also there are three nested loops to be run for each time the function is called. Especially the former, setting the standard cell to the midpoint makes the robot shift around the horizontal symmetry axis line by line, which significantly increases distance it needs to travel between two destinations and thus the time used to explore the map. The total time used and the map coverage is averagely around 1500 seconds (25 minutes) and 78%.

2.3 Our Implementation

We chose the wave front detection algorithm to implement. For choosing the next point, it will pick the cell with the smallest distance from the search start cell, which is the current robot position. (See the key code in Fig. 5 and Fig. 6).

Specifically, we implement the wave front detection using two nested breadth first search referencing from the suggested paper 'Frontier Based Exploration for Autonomous Robot' [1]. The outer one goes through the normal cells starting from the current robot position and checks if the current cell is a frontier or not. If the current cell is a frontier and is not visited yet, it enters the inner loop and adds the cell into the frontier list if the cell. At the end of both loops, the valid unvisited neighbours (the 'valid' means it is not out of boundary) of the current (frontier) cell will be pushed into the waiting list. It is notable that the neighbours of a frontier cell are much more likely to be frontiers.To get the closest frontier from the current robot position, we first calculate the distance and store the information into a list *fronterDisInfo* along with the cell coordinate, and then use the list comprehensions. Due to limitation of the *min()*, we add an exception to handle the error occurred when there is no cell in the list after removing the frontier cells that are too close to the robot.

As a result, the new algorithm takes around 30 minutes to explores the map with 78.24 % coverage.

```
# init
currentCell = searchStartCell
waitingList = [searchStartCell]
visitedList = []

while len(waitingList) != 0:

    currentCell = waitingList.pop(-1)

    if currentCell in visitedList or currentCell in self.blackList:
        continue

    # found a frontier
    if self.isFrontierCell(currentCell[0], currentCell[1]):

        # init to trace along the frontiers
        currentPotentialFrontier = currentCell
        waitingPotentialFrontierList = [currentCell]

        while len(waitingPotentialFrontierList) != 0:
            currentPotentialFrontier = waitingPotentialFrontierList.pop(-1)

            if currentPotentialFrontier in visitedList or \
                currentPotentialFrontier in self.blackList:
                continue

            if self.isFrontierCell(currentPotentialFrontier[0], currentPotentialFrontier[1]):
                frontierList.append(currentPotentialFrontier)

                for neighbours in self.getNeighbours(currentPotentialFrontier):
                    if neighbours not in visitedList:
                        waitingPotentialFrontierList.append(neighbours)

            visitedList.append(currentPotentialFrontier)

    # add target neighbours into waiting list
    for neighbours in self.getNeighbours(currentCell):
        if neighbours not in waitingList and neighbours not in visitedList \
                            and self.isInBoundary(neighbours):
            waitingList.append(neighbours)

    visitedList.append(currentCell)
```

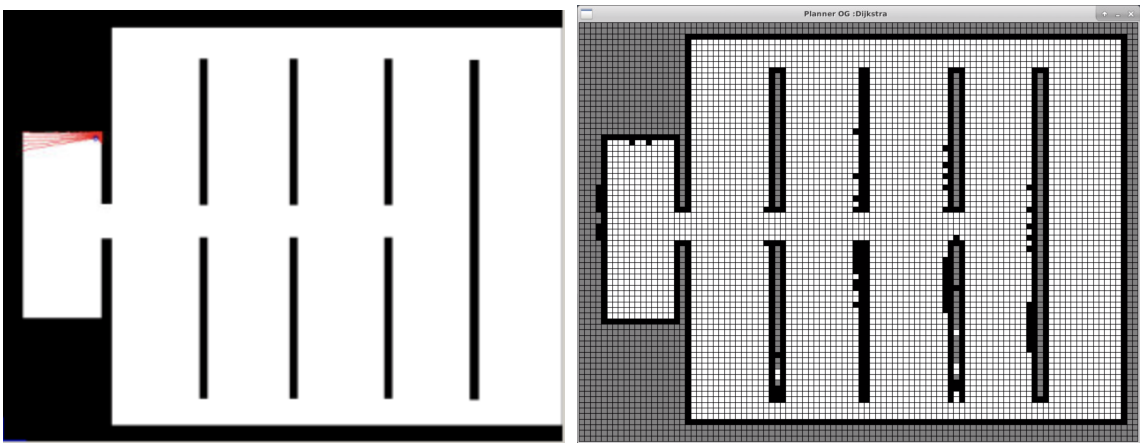
Fig. 5. Wave Front Detection

```
frontierDisInfo = self.getFrontiersDisInfo(frontierList)

try:
    candidate = [cell for dis,cell in frontierDisInfo \
        if dis == min(filter(lambda d : d > 0.5, [dis for dis,cellCoords in frontierDisInfo]))]
except ValueError:
    return []

return candidate[0]
```

Fig. 6. Closest Heuristic

3 Integration of Our Planner and Exploration System

See Fig.7 for the result. The coverage is 78.31%.



(a) STDR GUI (b) Planner Occupancy Grid

(c) Planner Search Grid (d) Mapper Node Occupancy Grid

Fig. 7. Result on the second launch script

4 Information-Theoretic Path Planning

4.1 Definition

In terms of path planning, the information theoritic approach picks the next waypoint which can reduce the uncertainty of map the most. One approach is using entropy, which based on the thermodynamic concept of irregularity, to assess the uncertainty. And to decide where to go, mutual entropy is used. In detail, it first pick a location and predict the observation. If it saw the observation, then update the map and check the effect on the map entropy. In our model where cells are either empty (0), unknown (0.5) and blocked (1), the entropy of a single cell is:

$$H(C) = -\sum p(c = c) ln(c = c) \tag{1}$$
$$= -(1 - p(c)) ln(1 - p(c)) - p(c) ln\, p(c) \tag{2}$$
$$\tag{3}$$

And the mutual information is given by:

$$I(x) = H(C) - H(C|x) \tag{4}$$

Where H(C|x) is the expected entropy such that

$$H(C|x) = \sum_{C} p_{C_k}(M) H(C|x, z(M)) \tag{5}$$

Where x, z is the destination and observation

4.2 Our Modification

In our model, we simply use the following equation to compute the entropy of the map once every 5 seconds of simulation time (the code is shown in Fig.8).

$$p_c(M) = |C_U| ln(2) \tag{6}$$

```
def calculateAndOutputEntropy(self):
    l = [[x,y] for x in range(0, self.occupancyGrid.getWidthInCells()) \
             for y in range(0,self.occupancyGrid.getHeightInCells())]
    unknownCellNum = len(filter(lambda xs: self.occupancyGrid.getCell(xs[0],xs[1]) == 0.5,l))

    entropy = unknownCellNum * np.log(2)

    # write into file
    f = open('entropy_data_baseline.txt','a+')
    print >> f, str(entropy) + '\n'
    rospy.loginfo("\n\n\n\noutput entropy" + str(entropy))
    f.close()
```

(a)

```
def run(self):
    #init
    f = open('entropy_data_baseline.txt','w')
    f.close()
    startT = rospy.get_time()

    while not rospy.is_shutdown():
        if (rospy.get_time() - startT) > 5.0:
            startT = rospy.get_time()
            rospy.loginfo("5s pass, outpyt entropy")
            self.calculateAndOutputEntropy()
```

(b)

Fig. 8. Our Implementation

4.3 Performance Evaluation

As shown in the graph (Fig.9), the two approach give similar performance with after 300 time steps (i.e, 1500 seconds after starting) the baseline algorithm outweigh the others, though theoretically the wave front detection should have a better performance. The total time to explore the map used by the two approaches (mentioned in the previous sections) also support that. We think the reason may be due to that the way we implement the detection approach makes the searching frontiers phase take too much time. Besides this main reason, during the exploration, we found that in most cases the robot will choose to go out of the initial room first instead of exploring the room corner. Then the robot has to re-enter the room later. In addition, we found that, since it will always pick the closest frontier, and whenever the robot reaches the destination, the search frontier phase will be called. As a result, even if the robot has not moved forward for a significant distance, the search algorithm has been called multiple times, each time with a non-negligible cost. Based on our situation, we will suggest that the baseline algorithm is more effective.

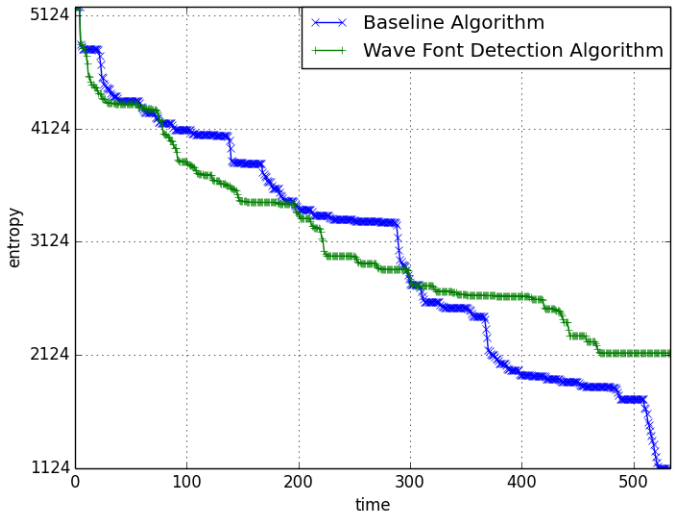


Fig. 9

References

[1] Anirudh Topiwala; Pranav Inani; Abhishek Kathpal (2018) Frontier Based Exploration for Autonomous Robot <*https://arxiv.org/abs/1806.03581*>.

[2] Brian Yamauchi (1997) A Frontier-Based Approach for Autonomous Exploration <*https://www.semanticscholar.org/paper/A-frontier-based-approach-for-autonomous-Yamauchi/a1875055e9c526cbdc7abb161959d76d14b58266*>.

[3] Callum Rhodes; Cunjia Liu; Wen-Hua Chen (2019) An Information Theoretic Approach to Path Planning for Frontier Exploration <*https://www.researchgate.net/publication/331929185_An_Information_Theoretic_Approach_to_Path_Planning_for_Frontier_Explor*

[4] Steven M. LaValle (2006) Planning Algorithm <*http://planning.cs.uiuc.edu*>.

[5] Matan Keidar; Gal A. Kaminka Efficient Frontier Detection for Robot Exploration Volume: 33 issue: 2 page(s):215-236 First published online: October 22, 2013 Issue published: February 1, 2014

[6] Robert M. Gray (2013) Entropy and Information Theory <*https://ee.stanford.edu/g̃ray/it.pdf*>.